VŠB – Technická univerzita Ostrava

Fakulta elektrotechniky a informatiky

Katedra informatiky

Vytváření automatických testů modulů na platformě .NET

Automatic unit Testing Creation on .NET Platform

2013 Vojtěch Tomeš

VŠB - Technická univerzita Ostrava
Fakulta elektrotechniky a informatiky
Katedra informatiky

Zadání bakalářské práce

Student: **Vojtěch Tomeš**

Studijní program: B2647 Informační a komunikační technologie

Studijní obor: 2612R025 Informatika a výpočetní technika

Téma: Vytváření automatických testů modulů na platformě .NET
Automatic unit Testing Creation on .NET Platform

Zásady pro vypracování:

Cílem práce je vytvoření automatizovaných testů modulů (unit testing) pro systém PneuB2B, který slouží pro zprostředkování nákupu a prodeje mezi jednotlivými obchodníky, kteří se zabývají prodejem pneumatik a disků pro automobily.
Automatizované testy budou vytvářeny s ohledem na metriku, která vyjadřuje pokrytí kódu testy.
Postup pro vypracování:
1. Seznamte se problematikou automatického testování komponent a s metrikami pokrytí testy.
2. Seznamte se s programovou podporou pro vytváření automatizovaných testů pro jazyk C# (platformu .NET) a s měřícími nástroji, které umožňují automatické měření pokrytí kódu testy. Vytvořte přehled těchto nástrojů.
3. Vyberte vhodnou metriku pro měření pokrytí kódu testy (nejlépe branch coverage nebo path coverrage)
4. Navrhněte nastavení prostředí, které umožní snadné vytváření automatizovaných testů programátory.
5. Vytvořte automatizované testy, které pokrývají vybraný modul dle zvolené metriky na přijatelnou míru (zdůvodněte, že toto pokrytí je dostatečné).

Seznam doporučené odborné literatury:

[1] KANER, C., BACH, J. AND PETTICHORD, B. Lessons Learned in Software Testing. 1 ed.: Wiley, 2001. 352 p. ISBN 0471081124
[2] MCCAFFREY, J.D. Software Testing: Fundamental Principles and Essential Knowledge. BookSurge Publishing, 2009. 118 p. ISBN 1439229074
[3] PATTON, R. Software Testing. 2 ed.: Sams Publishing, 2005. 408 p. ISBN 0672327988

Formální náležitosti a rozsah bakalářské práce stanoví pokyny pro vypracování zveřejněné na webových stránkách fakulty.

Vedoucí bakalářské práce: **Ing. Jan Kožusznik, Ph.D.**

Datum zadání: 16.11.2012

Datum odevzdání: 07.05.2013

doc. Dr. Ing. Eduard Sojka
vedoucí katedry

prof. RNDr. Václav Snášel, CSc.
děkan fakulty

Prohlašuji, že jsem tuto bakalářskou práci vypracoval samostatně. Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

V Ostravě 7. Května 2013 .....

Abstrakt

Tato bakalářská práce je zaměřena na testování, především modulů, na platformě .NET. V první části bakalářské práce si přiblížíme všeobecně problematiku testování a zaměříme se na různé druhy testování, jejich význam a použití. V druhé části následují metriky používané pro určování pokrytí kódu testy. Ve třetí části jsou popsány možnosti tvorby testů a metriky pokrytí, které nabízí Visual Studio 2010 a add-iny, které nabízejí rozšíření v možnostech tvorby testů a měření pokrytí kódu testy. Tyto možnosti jsou shrnuty v přehledu. Ve čtvrté části následuje návrh nastavení vývojového prostředí, vybraných modulů poskytující nám možnosti měření pokrytí kódu námi zvolenou metrikou a nastavení všech potřebných atributů jako například testovací databáze pro co nejsnazší vytváření automatizovaných testů, je zde také shrnut postup otestování vybraného modulu, dosažené pokrytí kódu a odůvodnění, zda a proč je dosažené pokrytí dostatečné.

Klíčová slova: test; metriky pokrytí kódu; Visual Studio 2010; add-in; platforma .NET.

Abstract

This bachelor thesis is focused on testing, primarily unit testing on .NET Platform. First part zooms in on testing in general and then focuses on different types of testing, their meaning and usage. In the second part follows metrics used in testing to determine code coverage. In the third part are described possibilities for testing and code coverage in Visual Studio 2010 and add-ins which offer new features in test creation and code coverage. These features are presented in summary. In fourth part follows set-up of development environment, chosen modules offering us possibility to measure code coverage with chosen metric and setting of all needed attributes like testing database for the easiest creation of automated tests, summarized process of testing chosen module, attained code coverage and justification if and why is attained coverage sufficient.

Key words: test; code coverage metrics; Visual Studio 2010; add-in; .NET Platform.

Obsah

Úvod

Testování je důležitou částí vývoje softwaru, zvláště v dnešní době, kdy se klade čím dál větší důraz na kvalitu. Pro zaručení této kvality se nám nabízí široké spektrum možných testů a nástrojů zpřístupňující nám tyto možnosti na různých platformách.

Důležitou částí je taktéž schopnost určit, jak je námi vyvíjený software kvalitní a být schopni tuto kvalitu doložit, určit postup ve vývoji, předcházet budoucím problémům a tím ušetřit čas i zdroje, které by bez této prevence bylo nutno vynaložit.

Cílem této bakalářské práce je zmapovat různé druhy testů, zjistit jejich využití v rozdílných etapách vývoje a situacích a určit nástroje, které pomáhají v tvorbě testů. Poté toto snažení určit metrikami pokrytí kódu, kterých je taktéž více druhů a lze je využívat v některých případech i současně.

V první části se seznámíme s testováním obecně a zaměříme se na teoretický popis jednotlivých druhů a typů testů, jejich slabiny i silná místa a etapy vývoje, kde se tyto testy využívají.

V další části se dozvíme, k čemu slouží pokrytí kódu, a seznámíme se s několika druhy pokrytí kódu testy, jejich slabinami i silnými stránkami a rozdíly mezi jednotlivými druhy pokrytí.

V posledních částech si představíme některé frameworky a nástroje s nimi spojené, které nám umožňují snáze tvořit unit testy a přidávají různé možnosti. Také si projdeme způsob vytvoření testu, zvolíme si požadované metriky a určíme hranici pokrytí, kterou budeme vyžadovat pro zvolený modul a zhodnotíme jeho pokrytí.

1. Testování, způsoby a přístup k testování a druhy testů

1.1 Testování

Testování software je pátrání za účelem zjištění kvality produktu nebo služby. Také nám poskytuje objektivní, nezávislý pohled na software, čímž nám umožňuje si uvědomit rizika vývoje software [3]. Testování software by se dalo definovat jako proces validace a ověřování, že produkt/služba splňuje požadavky které, byly použity pro design a vývoj a pracuje, jak očekáváme [1], [6].

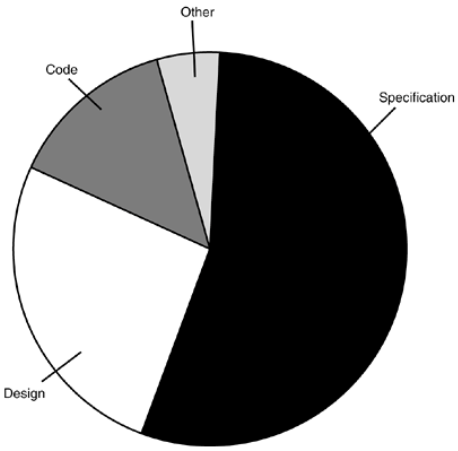
Testování software může být implementováno v jakékoliv fázi vývoje softwaru, závisí hlavně na metodě testování, kterou použijeme. Tradičně většina testování probíhala až po definici požadavků a dokončení psaní kódu [4]. Na druhou stranu, v agilních přístupech probíhá většina testování souběžně s vývojem, z čehož plyne, že metodologie testování je závislá na zvolené metodologii vývoje. Různé modely vývoje software budou zaměřeny na testy v různých částech vývoje software [8]. Příkladem může být z agilních metod Test-driven Developement, které dává velkou část testů do rukou vývojářům, ještě než se software dostane k týmu testerů.

Testování nikdy neodhalí všechny defekty a chyby v software, na druhou stranu nám dává nástroj ke kritice a porovnání současného stavu a chování proti oracle, což je pravidlo nebo mechanismus díky kterému lze rozeznat problém. Oracle může být například specifikace, kontrakt, starší verze produktu, srovnatelné produkty, uživatelské nebo zákazníkovy očekávání, platné standardy, zákony a další kritéria [9].

Hlavním cílem testování je zjišťování selhání software, díky čemuž může následně dojít k nalezení a odstranění chyby. Testování nám nemůže zaručit úplnou funkčnost produktu za jakýchkoliv podmínek, může nám pouze dát informaci, že software nefunguje tak, jak jsme za daných podmínek očekávali [1]. Rozsah testování většinou zahrnuje zkoumání kódu, spouštění software za různých podmínek a v rozdílných prostředích a zkoumání aspektů kódu, jestli kód dělá co má a co musí dělat. Každý software má svoji cílovou skupinu, na kterou je zaměřen, a z toho plynou také různé požadavky na kvalitu. Vývoj počítačové hry má zcela jistě jinou cílovou skupinu než například bankovní software, také požadavky na kvalitu a bezchybnost budou jiné mezi počítačovou hrou a software pro řízení raketoplánu, kde bude muset být otestováno před ostrým provozem všechno.

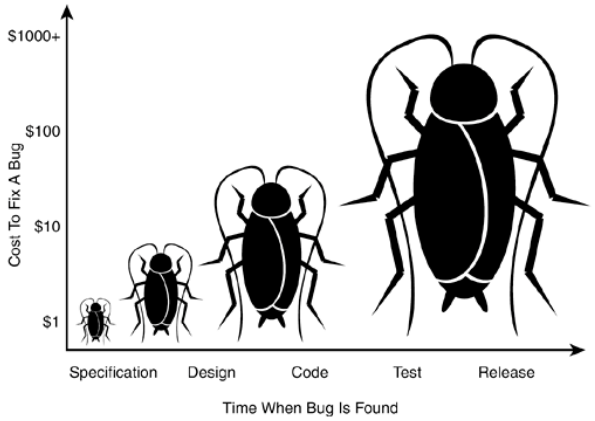
Ne všechny defekty software jsou způsobeny chybami v kódu. Častějším problémem je mezera ve specifikaci nebo špatný návrh. Běžným zdrojem těchto problémů bývají nefunkcionální požadavky jako testovatelnost, udržovatelnost, výkon, bezpečnost atd. [3]. V případě chyby v kódu může nastat při spuštění software k selhání nebo dostaneme špatné výsledky. Taková

chyba se nemusí projevit vždy a navíc může jedna takováto chyba způsobit více symptomů selhání.



Základním problémem v testování je, že provést testování na všechny možné kombinace vstupů je nereálné, i u relativně malých projektů[1]. To znamená, že množství defektů v software může být velký a defekty, které se projevují velmi málo, jsou obtížně odhalitelné při testování. Navíc u nefunkčních požadavků jako použitelnost, spolehlivost, výkon a kompatibilita je hranice kvality subjektivní [13].

Význam testování je samozřejmě i ekonomický. Čím později odhalíme závažné nedostatky a chyby, tím dražší je jejich následná náprava, zvláště chyby ve specifikaci a návrhu [4]. Tento nárůst v ceně opravy chyby je logaritmický, v raném stádiu vývoje taková náprava téměř nic nestojí, není li však taková chyba včas odhalena a dostane se až do produkce, kde ji odhalí zákazník, může taková oprava stát i tisíckrát více, než by stála, kdyby se odhalil nedostatek včas [12].



1.2 Způsoby a přístup k testování

1.2.1 Statické a dynamické testování

Statické testování je testování, při kterém se software nespouští. Obvykle se nejedná o detailní testování, jde především o manuální kontrolu a procházení kódu a dokumentů za účelem nalezení chyb [1]. Tento typ testování je dobře použitelný vývojářem, který psal kód. Statické testování také zahrnuje revizi požadavků a specifikací s ohledem na úplnost a vhodnost pro aktuální úkol. Jedná se o verifikaci z procesu verifikace a validace software. I statické testování lze automatizovat [12]. Takováto sada testů se skládá z interpretů nebo překladačů, které vyhodnocují syntaktickou správnost.

Dynamické testování zkoumá proměnné chování kódu. Jedná se o analýzu odezvy systému na proměnné, které se mění v čase. Dynamické testování probíhá na rozdíl od statického na software, který musel být zkompilován a spuštěn. Testování zahrnuje vkládání vstupů a kontroly, jestli výstup, který dostaneme, odpovídá předpokládaným výsledkům. Mezi dynamické testy patří například unitové, integrační a akceptační testy. Dynamické testování se ve většině případů automatizuje, ale lze jej provádět i manuálně [5].

1.2.2 Bílá skříňka

Je to metoda testování software, která testuje vnitřní strukturu a funkce aplikace. Při návrhu testovacích případů pro tento typ testování se používá vnitřní perspektiva systému. Tester volí vstupy tak, aby prošli určitými cestami v kódu, a určí přijatelné výstupy [7], [13]. Tento způsob testování se využívá především v unit testech, bývá však využíván i při integračních a systémových testech [6]. Tímto způsobem se dají dobře odhalit chyby a problémy, nemusí však odhalit chybějící části specifikace nebo nedostatečně popsané požadavky. Cílem tohoto typu testování je vědět, které řádky kódu jsou prováděny a schopnost identifikovat co by mělo být správným výstupem [7].

Základní postup při testování jako bílá skříňka zahrnuje hluboké porozumění kódu, který se testuje. Programátor musí znát dobře aplikaci, aby byl schopen navrhnout takové testovací případy, které by vykonali všechny viditelné cesty v kódu. Jakmile jsou tyto podmínky splněny, může dojít k analýze testovacích případů a jejich vytvoření [7], [13]. Vytvoření testovacího případu probíhá ve třech základních krocích:

1) Vstup - zahrnuje různé druhy požadavků, funkčních specifikací, dokumentů, bezpečnostních specifikací a zdrojový kód. Toto je přípravná fáze, ve které se shromáždí a uspořádají všechny základní informace [9].

2) Zpracování – zahrnuje navržení testovacího plánu, analýzu rizik a spuštění testovacích případů. V této části se ujišťujeme, že testovací případy důkladně testují aplikaci a dochází ke správnému zaznamenávání výsledků [9].
3) Výstup – finální zpráva zahrnující všechny přípravy a výsledky testů.

Tento typ testování má své výhody jako optimalizace díky nalezení skrytých chyb, introspekce jelikož programátor podrobně popisuje každou novou implementaci, také to napomáhá refactoringu, jelikož si můžeme pomocí již existujících testů ověřit, že jsme refactoringem nezanesli do kódu nové chyby [3]. Má však také své nevýhody a to především díky komplexnosti testování vyžadující rozsáhlé znalosti programu, aby bylo možné otestovat každou důležitou část, a v některých případech není možné provést test na všechny existující podmínky, a tudíž zůstanou některé části neotestované.

1.2.3 Černá skříňka

Jedná se o metodu testování, která zkoumá funkčnost aplikace bez bližší znalosti její vnitřní struktury nebo fungování. Tento způsob testování se dá aplikovat prakticky na každé úrovni testování software. Tester ví pouze, co by měl software dělat, neví už však jak. Bližší znalost kódu a vnitřní struktury není vyžadována [2], [12]. Tato vlastnost je jednou z výhod, jelikož tester samotný nepotřebuje programátorské znalosti a takovýto tester může mít na problém jiný náhled než programátor, čímž se může zaměřit na jiné části funkcionality. Na druhé straně může být tato neznalost i nevýhodou, jelikož můžeme vytvořit spoustu testovacích scénářů, které ve skutečnosti testují část, která by se dala otestovat snadno jedním testem nebo může nějaké části úplně vynechat [7].

Testovací případy jsou vybudovány kolem specifikace a požadavků. Z nich zjistíme, co má aplikace dělat, jaké vstupy jsou validní a které ne a jakými výstupy by měla reagovat a podle těchto informací vytvoříme testovací případy. Tímto způsobem testujeme především funkcionální požadavky, ale dají se tak testovat i nefunkcionální požadavky [4].

1.2.4 Šedá skříňka

Jedná se o kombinaci bílé a černé skříňky. Cílem tohoto testování je nalezení nedostatků jak ve struktuře, tak i v nesprávném použití aplikace. V šedé skříňce zná tester částečně vnitřní strukturu aplikace včetně dokumentace, vnitřní datové struktury a použitých algoritmů [13]. Díky znalosti kódu můžou testeři vytvářet lepší testy, i když jej testují z vnějšku. Můžeme tak předejít problému z černé skříňky, kdy byla jedna část kódu zbytečně testována spoustou testů. Mezi další výhody patří, že jsou tyto testy založeny na specifikaci a architektuře a nikoliv na zdrojovém kódu, i přes znalost vnitřního fungování zachovává hranici mezi testerem a vývojářem [7]. Toto testování má ale také své nevýhody, například neúplný průchod všech cest v kódu kvůli neúplnému přístupu ke zdrojovým kódům a těžká identifikace problémového

místa v distribuovaných systémech [1]. Přesto se tento typ testování dobře hodí na testování webových aplikací, jelikož není možné použít bílou skříňku z důvodu chybějících zdrojových kódů a binárních souborů. Také se dobře hodí pro funkční testování a testování business vrstvy jelikož se ve funkčním testování jedná prakticky o interakci uživatele se systémem a napomáhá to i v potvrzení, že software splňuje definované požadavky.

1.2.5 Testování zdola nahoru a shora dolů

Testování zdola nahoru je přístup k testování, kde se nejdříve testují základní komponenty jako moduly, procedury a funkce. Ty se integrují do modulů na vyšší úrovni, které se znovu testují. Tento proces se opakuje, dokud nevzniknou a nejsou otestovány komponenty na vrcholu hierarchie [12]. Tento přístup je použitelný pouze, pokud jsou všechny nebo alespoň většina modulů na stejné úrovni je připravena. Tato metoda také napomáhá v určení stádia vývoje a usnadňuje nám vyjádření postupu ve vývoji.

Testování shora dolů je opačný proces, kde jsou nejdříve testovány integrované moduly na nejvyšší úrovni a poté se testují jednotlivé větve modulu až po nejzákladnější části. Chybějící části jsou nahrazeny dočasnými moduly, které nemívají potřebnou funkčnost, ale mohou sloužit jako dočasná náhražka pro potřeby integrace a testování [12].

1.3 Druhy testů

1.3.1 Unit testy

Unit test je způsob testování, který testuje individuální části kódu tzv. unity. Jedná se o sady jednoho nebo více programových modulů s příslušnými kontrolními daty a operačními procedurami, které jsou testovány, k zjištění zda jsou vhodné k použití [3]. Intuitivně se dá za unit označit nejmenší testovatelná část aplikace. V procedurálním programování se za unit může považovat celý modul, běžněji se však jedná o jednotlivou funkci nebo proceduru [3]. V objektově orientovaném programování se nejčastěji za unit považuje celý interface jako například třída, ale může to být i jednotlivá metoda [8]. Tento typ testů běžně vytváří programátoři přímo během vývoje.

V ideálním případě jsou jednotlivé testovací případy na sobě nezávislé a při testování se využívá náhrad za chybějící části, aby bylo možno testovat moduly v izolaci. Unit testy píše běžně přímo vývojář, aby se ujistil, že kód odpovídá designu a chová se, jak se od něj očekává. Implementace může být různá od zcela manuální až po formalizování jako součást automatického buildu [9].

Cílem unit testů je izolovat jednotlivé části programu a ukázat, že jsou tyto části v pořádku [3]. Unit test je v podstatě prostředek, který nám popisuje, jak se část kódu musí chovat a díky čemuž získáme několik výhod.

První z výhod je schopnost odhalit problémy brzo ve vývojovém cyklu. V test-driven developementu, který je často využíván u agilních metodologií jako extrémní programování a scrum se unit testy vytvářejí ještě před tím, než se napíše samotný kód. Ve chvíli, kdy takovýto test proběhne úspěšně, považuje se napsaný kód za úplný [6]. Stejné unit testy se spouštějí opakovaně s tím, jak se kód mění a rozšiřuje se. Tento proces se může provádět buďto s každou změnou v kódu nebo automaticky při buildu kódu. Pokud test selže, považuje se to za chybu ve změněné části kódu nebo v testu samotném. Unit testy umožňují snadné dohledání chyby. Jelikož jsou takovéto chyby nalezeny ještě dříve, než je kód předán testerům nebo zákazníkovi, je odstranění těchto chyb poměrně snadné a levné, jelikož je to stále v rané části vývojového procesu [9].

Další nespornou výhodou je umožnění pozdějšího refactoringu kódu a ujištění, že modul funguje stále správně. Nejlepším postupem je napsat testy pro každou funkci a metodu, aby bylo možné rychle identifikovat chyby vzniklé jakoukoliv změnou a následně je opravit. Pokud máme takto neustále přístupné testy, není pro vývojáře problém kdykoliv zkontrolovat, zda část kódu funguje tak jak má. V prostředích s neustálým unit testováním nám tyto testy umožní přesný náhled na zamýšlené použití spustitelného kódu navzdory jakýmkoliv změnám. Při dobrém pokrytí testy a praktikách při vývoji lze udržet dobrou funkčnost navzdory častým změnám či optimalizacím.

Unit testy mohou také snížit nejistotu v jednotlivých částech kódu a můžou být použity ve způsobu testování zdola nahoru. Díky testování jednotlivých částí a až poté jejich spojení značně usnadňuje integrační testování. Nelze však nahradit integrační testování složitou hierarchií unit testů.

Unit testy lze také považovat za jakýsi druh živé dokumentace systému. Vývojáři, kteří se snaží zjistit, jaká funkcionalita je nabízena nějakou částí a jak ji použít můžou snadno získat základní přehled tím, že se podívají na unit testy této části. Unitové testovací případy charakterizují kritické body, které jsou důležité pro úspěšný běh unit. Tyto charakteristiky zahrnují vhodné a nevhodné použití unit a také špatné chování, které by mělo být unitou zachyceno. Unitový testovací případ zachycuje takovéto kritické charakteristiky a dokumentuje je. Na rozdíl od toho se může běžná textová dokumentace snadněji vzdálit od skutečné implementace programu z důvodu změn v návrhu a laxnímu přístupu k udržování dokumentace aktuální [7].

Pokud je software vyvíjen pomocí test-driven developementu můžou unit testy zaujmout pozici formálního návrhu. Na každý unit test může být nahlíženo jako na element návrhu specifikujícího třídy, metody a pozorovatelné chování [12].

Pro lepší představu takovéhoto chování lze ukázat příklad

```
public class TestAdder {
    public void testSum() {
        Adder adder = new AdderImpl();
        // can it add positive numbers?
        assert(adder.add(1, 1) == 2);
        assert(adder.add(1, 2) == 3);
        assert(adder.add(2, 2) == 4);
        // is zero neutral?
        assert(adder.add(0, 0) == 0);
        // can it add negative numbers?
        assert(adder.add(-1, -2) == -3);
        // can it add a positive and a negative?
        assert(adder.add(-1, 1) == 0);
        // how about larger numbers?
        assert(adder.add(1234, 988) == 2222);
    }
}
```

Toto je příklad testovací třídy. Nejdříve musíme mít interface nazvaný Adder a implementující třídu s konstruktorem bez argumentů nazvaným AdderImpl. Následující assert nám udává, že musí obsahovat metodu add se dvěma integery jako vstupem a vracející integer. Specifikuje také chování metody pro malý rozsah hodnot. Jelikož byl tento unit test napsán první slouží jako návrh specifikující požadované řešení, ale implementační detaily jsou ponechány na programátorovi. Pokud použijeme praktiku o vytvoření co nejjednoduššího funkčního řešení tak dostaneme takovouto implementaci.

```
interface Adder {
    int add(int a, int b);
}
class AdderImpl implements Adder {
    int add(int a, int b) {
        return a + b;
    }
}
```

Na rozdíl od návrhů založených na diagramech má použití unit testu jako návrhu jednu velkou výhodu. Návrh jakožto unit test může být použit pro ověření, že se implementace řídí návrhem. S touto metodikou návrhu test nemůže projít, pokud vývojář neimplementuje řešení podle návrhu. Na druhou stranu je pravda, že unit testy nejsou tak dobře přístupné a čitelné jako diagramy, nicméně v dnešní době existují nástroje, které jsou schopné vygenerovat UML diagramy pro většinu moderních jazyků.

Často se také stává, že mají některé třídy reference na jiné třídy a testování takovéto třídy může přetéct do třídy, na kterou odkazuje. Běžným příkladem může být třída závislá na databázi. Aby se otestovala taková třída, napíše tester kód, který interaguje s databází. Toto je ovšem špatně a unit test by neměl překročit hranice své třídy a tím spíše by neměla překročit takovou hranici mezi procesem a sítí, jelikož by to mohlo způsobit nepřijatelné problémy s výkonem testování.

Pokud bychom překročili takovéto hranice, stal by se z unit testu test integrační a při selhání takovéhoto testu by nebylo jednoznačné, která komponenta způsobila selhání testu. Místo toho by se měl vytvořit abstraktní interface kolem databázových dotazů a ten implementovat pomocí mock objektů. Díky abstrakci těchto potřebných vazeb z kódu může být nezávislý unit důkladněji otestován, čímž dosáhneme vyšší kvality a lepší udržovatelnosti.

Existují i parametrizované unit testy, které na rozdíl od tradičních, které jsou uzavřené metody, můžou přijímat sadu parametrů. Vhodné parametry můžou být dodávány ručně nebo v některých případech automaticky generovány testovacím frameworkem [7].

Testování však nezachytí všechny chyby, jelikož nelze vyhodnotit všechny cesty s výjimkou těch nejtriviálnějších programů. Unit testy na tom nejsou jinak. Navíc unit testy z definice testují pouze funkcionalitu unit samotných, z toho důvodu nezachytí integrační chyby a problémy na širším spektru jako například ve funkcích probíhajících nad více unitami nebo nefunkční oblasti testování jako výkon. Unit testování by se mělo provádět ve spojení s dalšími typy testů, jelikož jsou schopny poukázat na výskyt či absenci určitého druhu chyb, nelze jimi však dokázat celková absence chyb. K prokázání správného chování aplikace pro každou cestu a možný vstup a zajištění bezchybnosti jsou zapotřebí další způsoby testování.

Testování je v podstatě kombinatorický problém a například pro každé rozhodování pomocí boolean je zapotřebí dvou testů, jeden pro true a druhý pro false větev. Výsledkem tohoto je, že na každý řádek kódu je zapotřebí 3-5 řádků testovacího kódu. Toto je samozřejmě časově i finančně náročné a nemusí se takováto snaha vždy vyplatit. Existuje také spousta problémů, které se nedají snadno otestovat, jako například nedeterministické problémy či více-vláknové. Navíc kód psaný pro testy bude v sobě taktéž obsahovat chyby.

Další věcí potřebnou pro vytvoření realistických a užitečných unit testů je vytvoření relevantních počátečních podmínek tak, aby se testovaná část systému chovala jako část systému. Pokud by tyto počáteční podmínky nebyly nastaveny správně tak by test neprováděl kód v realistickém kontextu a tudíž by tím byla znehodnocen test a přesnost výsledků [8].

Abychom získali všechny dostupné výhody unit testů, je zapotřebí dodržovat přísná pravidla během vývoje. Je důležité udržovat si záznamy nejenom o všech provedených testech, ale také o všech změnách provedených ve zdrojovém kódu unit. Z toho plyne, že je velmi důležité použití verzovacího systému. Pokud nějaká nová verze unity selže v testu, který byl předtím úspěšný, může nám takovýto verzovací systém nabídnout seznam změn v kódu, které se staly mezi těmito dvěma testy [7].

Za tohoto důvodu je také důležité, aby všechny testovací případy, které selhaly, byly kontrolovány každý den a okamžitě řešeny. Pokud tento proces nebude probíhat a nebude integrován do práce týmu, nebude synchronizován vývoj aplikace a testovací prostředí což sníží efektivitu a zvýší možnost nekorektních výsledků z testování.

Další výzvou v unit testování jsou vestavěné systémy. Je to z důvodu, že je software vyvíjen v jiném prostředí, než v jakém poběží ve výsledku. Z toho důvodu není možné jednoduše testovat program ve skutečném prostředí, ve kterém bude nasazen, jako je tomu u desktopových aplikací [10].

Extrémní programování je příkladem, kde je unit testování použito jako jeden z jeho základních stavebních kamenů a závisí na automatizovaném frameworku pro unit testování. Extrémní programování využívá unit testů pro test-driven developement. Vývojář napíše test, který udává požadavek nebo nepřípustný vstup. Takovýto test musí selhat, protože požadavek ještě není naimplementován nebo test záměrně odhaluje existující chybu v kódu. Poté vývojář napíše nejjednodušší možný kód tak, aby byl test úspěšný. Většina kódu je takto pokryta unit testy, není však pravdou, že by museli být pokryty zcela všechny cesty kódem [5]. Extrémní programování prosazuje strategii „otestuj vše, co by se mohlo rozbít“ místo tradičního „otestuj všechny možné cesty, co existují“. Toto vede k vývoji méně testů než u klasických metod. Tímto přístupem extrémní programování uznává, že testování se málo kdy provádí kompletně, jelikož by takovéto snažení zabralo spoustu času a prostředků než aby se vyplatilo, a poskytuje návod, jak efektivně využít omezených zdrojů.

Klíčové je také, aby se testovací kód udržoval na stejné úrovni kvality, jako implementace samotná. Vývojáři odesílají testovací kód do repositářů spolu s testovaným kódem. To umožňuje získání všech dříve zmíněných výhod, jako jednodušší a jistější vývoj kódu, usnadnění refactoringu, jednodušší integrace, modulárnější design a slouží také jako přesná dokumentace. Také tím, že se spouští neustále dokola dochází k regresnímu testování.

Unit testy jsou běžně automatizované, ale nic nebrání tomu, aby byly prováděny manuálně [5]. Manuální přístup může zahrnovat instrukční dokument popisující krok po kroku postup. Nicméně je cílem unit testování izolovat jednotlivé unity a ověřit jejich správnost a nejefektivnějším přístupem je automatizace. Naopak špatně naplánované manuální testy unit testy můžou přejít do integračních testů, které zahrnují více komponent současně a tím zamezit většině cílů a výhod unit testování [7]. Abychom plně dosáhli efektu izolace za použití automatizovaného přístupu, je testovaná část kódu spouštěna ve frameworku mimo prostředí produktu a kontextu volání, pro který byl původně vytvořen. Testování v takovémto prostření nám může odhalit nepotřebné závislosti mezi testovaným kódem a ostatními unitami, které mohou být následně odstraněny [8]. Při použití automatizovaného frameworku napíše vývojář do testu kritéria, které ověřují správnost unity. Během průběhu testů framework zapíše všechny kritéria, které selhali a vytvoří zprávu. V případě že se jedná o závažnou chybu, může framework pozastavit vykonávání následujících testů. Důsledkem tohoto je motivace programátorů vytvářet oddělené a soudržné těla kódu, čímž podporují správné návyky ve vývoji software. Sloučením návrhových vzorů, unit testování a refaktoringu tak vznikají jedny z nejlepších řešení.

1.3.2 Integrační testy

Je to fáze testování, ve které jsou individuální softwarové moduly spojeny a testovány společně. Probíhá po unit testech, ale před validačními testy. Integrační testování vezme jednotlivé vstupní moduly, které byly otestovány jako unity a spojí je ve větší agregáty, aplikuje testy definované v integračních testových plánech a výsledkem je integrovaný systém, který je připraven pro celkové otestování [8].

Hlavní význam integračních testů spočívá v ověření funkčních požadavků, výkonu a spolehlivosti kladených na základní části návrhu. Těmito základními částmi jsou skupiny unit, které jsou spouštěny přes své interfacy za použití metody černé skříňky a jednotlivé případy jsou simulovány pomocí příslušných vstupních parametrů a dat [13]. Simulované použití společných dat a komunikace mezi procesy jsou testovány jednotlivé subsystémy. Testovací případy jsou konstruovány tak, aby zjistili, zda všechny komponenty takového subsystému spolu správně komunikují pomocí vzájemného volání procedur a aktivace procesů. Tento proces vytváří stavební kameny, které jsou po úspěšném otestování přidávány do již potvrzené části a slouží tak jako podpora integračních testů dalších částí. K integračnímu testování se dá přistupovat více způsoby, jako příklad poslouží testování zdola nahoru, shora dolů a metoda velkého třesku (Big Bang) [8].

V přístupu velkého třesku jsou všechny nebo alespoň většina modulů spojeny dohromady, aby vytvořili kompletní software nebo alespoň jeho velkou část a toto je pak použito při integračním testování. Tato metoda je časově velmi efektivní, ale pokud jsou testovací případy a jejich výsledky špatně zaznamenávány celý integrační proces se tím může značně zkomplikovat a zabránit tím dosažení cílů integračního testování. Jedním z druhů testování velkým třeskem je testování modelu použití. Tento typ testování spočívá ve spuštění zátěže, jak se očekává v reálném provozu v integrovaném uživatelském prostředí. Díky tomuto stylu testování můžeme dokázat funkčnost v prostředí a nepřímo tím také dokážeme správnost jednotlivých komponent jejich použitím. Tento přístup je značně optimistický, jelikož počítá pouze s malým množstvím problémů s jednotlivými komponentami, závisí tedy silně na tom, že vývojáři jednotlivých komponent individuálně provedly řádné unit testy nad jednotlivými komponentami. Cílem této strategie je neopakovat testy, které již provedli vývojáři a místo toho se zaměřit na problémy spojené se vzájemnou interakcí komponent v prostředí. Použití testování modelu užití v integračním testování může být efektivnější a nabízí lepší pokrytí testy než tradiční funkční integrační testy. Musí se ale klást velký důraz na správné určení zátěže, aby bylo možno vytvořit realistické scénáře pro testování a zaručili jsme tím, že integrovaný systém se bude chovat tak, jak jsme očekávali i u cílového zákazníka. Proto je důležité, aby se při specifikaci integračních testů nevynechali žádné podmínky, jelikož by systém pro takovéto opomenuté části nebyl otestován.

1.3.3 Akceptační testy

Akceptační testy je druh testů prováděný za účelem zjištění, zda jsou splněny požadavky specifikace nebo smlouvy. U software toto zahrnuje testování černou skříňkou na systému podle předem dohodnutých testovacích kritérií a případů [9]. Rozlišuje se také akceptační testování poskytovatelem systému od akceptačního testování u zákazníka, které určuje, zda bude produkt převzat.

Testování všeobecně zahrnuje spouštění sady testů nad kompletním systémem. Každý individuální test vykonává určitou funkci nebo vlastnost systému a výsledkem je úspěch nebo selhání testu [13]. Všeobecně se neurčuje žádný stupeň funkčnosti nebo nefunkčnosti. Testovací prostředí je běžně navrženo tak, aby bylo co nejvíce identické s předpokládaným prostředím, kde bude systém nasazen včetně extrémních situací. Každý testovací případ musí obsahovat vstupní data a formální popis, jaké operace se mají vykonat. Cílem je co nejjasněji popsat daný testovací případ a očekávané výsledky. Akceptační kritéria jsou nejčastěji vytvořeny zákazníkem a vyjádřeny doménově specifickým jazykem. Tyto testy jsou na vysoké úrovni a slouží k ověření úplnosti jednotlivých případů užití. Akceptační testy jsou poté z těchto kritérií vytvořeny nejlépe ve spolupráci mezi zákazníkem, analytikem, testery a vývojáři. Je důležité, aby obsahovali jak bisiness logiku tak i validaci UI. S tím, jak jsou plněny jednotlivé případy užití ze specifikace, se lze ujistit, zda se vývoj ubírá správným směrem.

Jednotlivé testy jdou ideálně vytvořeny na začátku každé iterace ještě před tím, než začne samotný vývoj, aby měli vývojáři představu co vyvíjet [1]. Někdy akceptační testy můžou obsahovat více případů, které jsou vyvíjeny v různých iteracích zároveň a na jejich testování se musí přistoupit jinak. Populárním řešením je vytvoření náhrad za externí interface nebo data tak, aby imitovali tyto chybějící případy užití. I přes to se nepovažují takovéto případy užití za kompletní, dokud neproběhl celý akceptační test bez náhrad.

Proces aplikačních testů probíhá tak, že je testovací sada spouštěna nad dodávanými vstupními daty nebo za použití skriptu, který vede testery. Poté jsou obdržené výsledky srovnány s očekávanými, a pokud se pro každý testovací případ výsledky shodují, dá se považovat tato testovací sada za úspěšnou [9]. Pokud ne, je systém zamítnut zákazníkem, nebo přijat za předem domluvených podmínek. Cílem je ujistit se, že dodávaný systém splňuje požadavky jak zákazníka, tak i cílových uživatelů. Akceptační proces slouží jako poslední kontrola kvality, kde můžou být odhaleny chyby, které se nepodařilo odhalit dříve. Hlavní význam je v tom, že jakmile úspěšně proběhne akceptační testování a všechna akceptační kritéria jsou splněna, musí zákazník produkt převzít a zaplatit. Tento proces se nazývá User acceptance testing (UAT) neboli akceptační testování u zákazníka.

UAT je proces, který nám obstará potvrzení, že systém splňuje všechny požadavky, na kterých bylo dohodnuto. Toto testování probíhá přímo před zákazníkem a testy samotné provádí

proškolený uživatel [12]. Tyto testy jsou vyvozeny ze smlouvy nebo ze specifikace požadavků. Testeři navrhnout formální testy a ohodnotí je podle závažnosti. V ideálním případě by neměl akceptační testy nemněl navrhovat stejný člověk jako testy integrační a systémové. UAT slouží jako potvrzení vyžadovaných funkcí a správného fungování systému, simulující reálné podmínky a použití. Pokud software funguje bez problémů za normálního použití, dá se proto předpokládat stejná úroveň kvality i při nasazení do produkce. Tyto testy prováděné většinou koncovým uživatelem se nezaměřují na nevýznamné chyby jako překlepy nebo kosmetické chyby ani na úplnou snahu o likvidaci systému jeho pádem. Takovéto chyby se snaží vývojáři identifikovat a odstranit během předchozích fází testování.

V extrémním programování však výraz akceptační testování odkazuje na funkční testování případů užití během implementační fáze. Zákazník specifikuje scénáře, které je potřeba otestovat, aby se dal případ užití prohlásit za správný. Jeden takový případ může mít jeden i více akceptačních testů, aby byla zajištěna správná funkcionalita. Jedná se o systém černé skříňky, každý test reprezentuje nějaké očekávané výstupy ze systému. Zákazník je zodpovědný za kontrolu správnosti akceptačních testů a určují, které testy mají nejvyšší prioritu. Také může být v tomto kontextu použito jako regresní testování před vydáním software do produkce. Případ užití se nedá považovat za kompletní, dokud neprojde akceptačními testy, to znamená, že se pro každou iteraci musí vytvořit nové testy, jinak se nedá zaznamenat žádný pokrok ve vývoji.

Kromě UAT jsou ještě i další druhy akceptačního testování, které mohou probíhat současně. Jedním z nich je Operační akceptační testování (OAT) také známé jako testování operační připravenosti. Toto zahrnuje kontroly, které proběhly nad systémem, které zajišťují, že procesy a procedury umožňují systému fungovat a dovolují údržbu. Toto může obsahovat i kontroly zálohovacích zařízení, procedur pro obnovu z kritických stavů, školení cílových uživatelů, údržbu a bezpečnostní procedury.

Dalším druhem je alfa a beta testování. Alfa testy probíhají u vývojářů a zahrnuje testování funkčního systému zaměstnanci vývojářské firmy před vypuštění mezi externí zákazníky. Beta testování probíhá u zákazníka a zahrnuje testování skupinou uživatelů na svých lokálních stanicích a poskytujících zpětnou vazbu než je systém vypuštěn i mezi ostatní zákazníky.

1.3.4 Regresní testy

Regresní testování je jakýkoliv typ testování software, který hledá nové chyby nebo regrese v existujících funkcionálních a nefunkcionálních oblastech systému po změnách jako vylepšení, opravy nebo změny v konfiguraci. Záměrem regresního testování je zajistit, že takovéto změny nevytvoří nové chyby [5]. Jedním z hlavních důvodů regresního testování je určení, jestli změna v jedné části software nějak neovlivní fungování ostatních částí. Běžné metody regresního testování zahrnují spouštění dříve úspěšných testů a kontroly, zda se chování systému

nezměnilo nebo se nevrátily některé z dříve opravených chyb. Regresní testování se dá provádět značně efektivně, pokud systematicky volíme pouze testy, které jsou potřeba, aby byla změna adekvátně pokryta.

Ukázalo se, že při opravách systému není neobvyklé zanesení nových chyb nebo vrácení některých již dříve opravených chyb zpátky do systému [12]. Takovéto znovuobjevení chyby může být způsobeno například špatnou kontrolou revizí a oprava dané chyby se ztratí. Často je také oprava vytvořena tak, že vyřeší problém jenom pro určité případy, pro které byl pozorován, nikoliv už však pro obecné případy které mohou nastat během životního cyklu software. Ve spoustě případů také oprava problému v jedné části nevyhnutelně způsobí chybu v jiné části a je také možné, že při změně návrhu nějaké vlastnosti se některé chyby, které se udělali v původním návrhu, zopakují. Proto se považuje za dobrou praktiku při nalezení a opravě chyby zaznamenat si test, který tuto chybu odhalil a následně jej pravidelně spouštět po změnách v systému [3]. Toto může být prováděno manuálně, častější je však využití automatizovaných testovacích nástrojů. Takovéto testovací nástroje jsou využívány nejen k automatickému spouštění testů po změnách, v některých projektech se také používá automatického spouštění všech regresních testů po určitém časovém intervalu a hlásí se jakékoliv selhání testu, což může naznačovat regresi nebo zastaralý test. Běžnou praktikou ovšem je spouštění regresních testů u malých projektů po každé kompilací nebo u větších v určitý čas, nejčastěji v noci každý den nebo jednou za týden [3]. Regresní testování je také nedílnou součástí extrémního programování. V této metodě jsou dokumenty s návrhem nahrazeny rozsáhlými automatizovanými tesy celého softwarového balíku pro každou fázi vývojového cyklu.

Dříve byly regresní testy prováděny týmem, který se staral o kvalitu software. Nicméně tento proces probíhal až po dokončení vývoje a chyby nalezené v této fázi jsou drahé na odstranění. Tento problém byl částečně vyřešen zvýšeným zaměřením na unit testy. I když vývojáři psali testovací případy i dříve, jednalo se o funkcionální testy či unit testy, které jenom ověřovali očekávané výsledky. V dnešní době se dbá na to, aby se zaměřovalo na unit testy a aby tyto testy obsahovali jak pozitivní, tak i negativní testovací případy [5].

Využití regresního testování nespočívá pouze v otestování správnosti programu, často se také využívá ke sledování kvality výstupu[3]. Například v designu překladače by mohly regresní testy sledovat velikost kódu, čas simulace a kompilace sady testovacích případů. Regresní testy by se také daly v širším pojetí kategorizovat jako funkcionální nebo unit testy. Funkcionální testy vykonávají celý program s různými vstupy, může se jednat o naskriptovanou sérii programových vstupů včetně možnosti automatizovaných mechanismů pro kontrolu pohybu myši a kliků. Jak funkcionální tak i unit testy bývají automatizované.

1.3.5 Fuzz testy

Jedná se o techniku testování, často plně automatizovanou nebo alespoň částečně automatizovanou, která dává na vstup programu špatná, neočekávaná nebo zcela náhodná data. Program je poté monitorován a odchytávají se všechny problémy, způsobující pády software, selhávání předpokladů v kódu nebo se hledají potencionální úniky paměti [15]. Běžně se také využívá k testování bezpečnostních problémů v systému. Nejčastějšími cíly fuzz testování jsou formáty souborů a síťové protokoly, ale může se jednat v podstatě o jakýkoliv vstup programu. Mezi zajímavé vstupy patří proměnné prostředí, akce klávesnice a myši a sekvence volání API. I věci, které se běžně nepovažují za vstup, můžou být fuzzovány, například obsah databáze, sdílená paměť či přesné prokládání vláken. Z hlediska bezpečnosti jsou nejzajímavější takové vstupy, které přesahují hranici vstupů, kterým věříme. Například fuzzování kódu uploadu souborů je daleko důležitější a řekne nám o bezpečnosti určitě více, než fuzzování parsování konfiguračních souborů, které jsou přístupné jenom určitým uživatelům.

Fuzz testování je nejčastěji využívá metodologii černé skříňky a bývá ve větších projektech. Nabízí vysoké přínosy za cenu jeho vytvoření. I když tato technika ukazuje pouze to, že nějaká náhodná část software je schopná fungovat i navzdory zcela náhodným vstupům a výjimkám jimi vyvolanými bez úplného pádu software než aby dokázal, že se software chová tak jak má [14]. Na druhou stranu nám toto nabízí jisté ujištění o celkové kvalitě software, než aby se jednalo o nástroj k hledání chyb, a zcela určitě nemůže nahradit úplné vyčerpávající testování a formální metody [15]. Jakožto hrubý nástroj k měření spolehlivosti je schopno fuzz testování určit, na které části programu je třeba se zaměřit formou auditu, statické analýzy kódu či částečnými úpravami.

Mimo testování na úplné pády systému je fuzz testování také používáno k nalezení selhání předpokladů a úniků paměti, které mohou u velkých aplikací představovat značné bezpečnostní riziko [15]. Jelikož fuzz testování často generuje neplatné vstupy, používá se také k testování rutin pro odchytávání a obsluhu chyb a výjimek. Toto je důležité především u aplikací, které nekontrolují své vstupy. Jednoduché fuzzování můžeme považovat za dobrý způsob, jak zautomatizovat negativní testování. Fuzzování také může odhalit některé problémy v korektnosti. Například k nalezení chyb spojených se špatnou serializací, může také odhalit neúmyslné rozdíly mezi dvěma verzemi programu nebo dvěma implementacemi stejné specifikace.

Fuzz testy se obecně dělí do dvou kategorií. Mutační testy, mutují existující vzorky dat a tím vytváří nová testovací data. Generační testy, definují nová testovací data podle modelu vstupů [14]. Nejjednodušší forma fuzzingu spočívá v posílání proudu náhodných bitů do software jako příkazy příkazového řádku, náhodně zmutovaných protokolových paketů nebo eventů. Tato technika se stále často a úspěšně používá k nalezení chyb v aplikacích s příkazovou řádkou, síťových protokolech nebo ve službách a aplikacích založených na GUI [14], [15]. Další běžnou

technikou snadnou na implementaci je mutování existujících vstupů z testovacího prostředí přehazováním bitů nebo náhodným přesouváním bloků v souboru, avšak nejúspěšnější fuzzery jsou založeny na detailním porozumění formátu nebo protokolu, který testujeme.

Toto porozumění může být založeno na specifikaci. Fuzzer založený na specifikaci zahrnuje napsání celého pole specifikací do nástroje a poté využití generování testů založených na technice modelového generování testů které prochází specifikací a přidává anomálie do obsažených dat, struktury, zpráv a ´posloupností. Tato technika chytrého fuzzování je také známá jako testování robustnosti, syntaxe, gramatiky a fault injectingu na vstup [14]. Můžeme také zvolit heuristický přístup. Takovéto fuzzery mohou generovat testovací případy úplně od začátku z ničeho nebo můžou mutovat existující příklady testovacích dat z testovacího prostředí nebo reálného světa. Můžou se také zaměřovat na validní nebo nevalidní vstupy, kde při převaze validních vstupů dochází často k odhalení skrytých chyb.

U protokolově založeného fuzzingu založeného na specifikaci jsou hlavními limitacemi především to, že nelze začít s testováním, dokud není specifikace značně vyspělá, jelikož je podmínkou pro napsání takového fuzzeru. Další limitací je, že spousta užitečných protokolů jsou proprietární nebo zahrnují proprietární rozšíření. Pokud jsou fuzz testy založeny pouze na specifikaci, bude pokrytí nových rozšíření či proprietárních protokolů velmi omezené nebo nebudou pokryty vůbec [15].

Fuzz testování můžeme kombinovat s dalšími testovacími technikami. Fuzzing spojený s metodou bílé skříňky využívá symbolického provádění a řešení podmínek. Evoluční fuzzing využívá zpětné vazby z heuristiky jako pokrytí kódu z modelu šedé skříňky či chování modelového útočníka z černé skříňky a tím efektivně automatizovat přístup k explorativnímu testování [7].

Důležitým aspektem je i schopnost izolace a reprodukce nalezených chyb. Redukce testovacích případů, abychom získaly co nejmenší nutný počet takových testovacích případů. Tato redukce může probíhat manuálně či za použití softwarových nástrojů, a většinou zahrnuje strategii, rozděl a panuj, kde jsou části testu po jednom odstraňovány, dokud nezůstane jenom základní jádro testovacích případů [7]. Co se schopnosti reprodukce chyb týče, fuzzovací software většinou zaznamenává vstupní data, které produkuje, ještě před aplikací na testovaný software. Pokud systém okamžitě spadne, jsou testovací data zachována. Pokud se ve fuzzingu využívá proudu generovaných pseudonáhodných čísel, používá se k reprodukci hodnota seedu. Jakmile je chyba nalezena některé fuzzovací softwary vytvoří testovací případ, který je použit při debuggingu nebo nejdříve projde redukcí testovacích případů.

Hlavním problémem fuzzingu spočívá v tom, že obecně odhaluje velmi jednoduché problémy. Výpočetní složitost softwarového testování je exponenciální řada a k nalezení něčeho zajímavého pomocí fuzzeru je zapotřebí využít zkratek. Primitivní fuzzer může mít velmi špatné

pokrytí kódu. Nástroje pro měření pokrytí kódu jsou používány k určení, jak dobře fuzzer funguje, jedná se ale pouze o jakýsi náznak kvality. Každý fuzzer obecně najde jinou sadu chyb.

Na druhou stranu se někdy díky fuzz testování podaří odhalit velmi závažné chyby nebo chyby zneužitelné reálnými útočníky. S tím jak se stalo fuzz testování známější je tento případ o to častější, jelikož se stejné techniky a nástroje jako pro testování využívají k útokům s cílem zneužít software [14], [15]. Toto je velká výhoda proti binárnímu nebo zdrojovému auditingu či fault injection, které závisí na umělých chybových podmínkách, které jsou těžce využitelné až v některých případech je nemožné je zneužít.

Náhodnost vstupů vytvářených při fuzz testování je ale často považováno za nevýhodu, jelikož se těžko určují podmínky hraničních hodnot v těchto náhodných vstupech. V dnešní době většina fuzzerů řeší tento problém využitím deterministických algoritmů založených na uživatelských vstupech. Fuzz testování vylepšuje bezpečnost a spolehlivost software, jelikož často objeví zvláštnosti a chyby, které by lidští testeři nenalezli nebo jsou snadno přehlédnutelné a i pečlivý návrh testů by je ve většině případů nezahrnul.

2. Pokrytí kódu a metriky pokrytí kódu

2.1 Pokrytí kódu

Analýza pokrytí kódu spočívá v hledání oblastí programu, které nejsou vykonávány žádným z testů ze sady testovacích případů, vytváření dalších testů, aby zvýšily pokrytí a odstranili tyto nepokrytá místa. Tím můžeme určit kvantitativní míru pokrytí kódu, což slouží jako nepřímá míra kvality [11]. Dále nám také může poukázat na redundance v testovacích scénářích, které nám nezvyšují míru pokrytí. Analyzátory pokrytí kódu nám pomáhají tento proces automatizovat.

Analýzu pokrytí využíváme k určení kvality nikoliv vlastního produktu, ale spíše naší sady testů. Tato analýza vyžaduje přístup ke zdrojovému kódu testovaného programu, proto se analýza pokrytí nedá provádět u testování metodou černé skříňky při testování dokončeného produktu [6]. Analýza pokrytí má své silné stránky ale také slabiny a je potřeba si vybrat správnou metodu měření pokrytí kódu. Také je důležité si určit minimální procentuální hranici pokrytí, abychom věděli, kdy můžeme s analýzou přestat.

Analýza pokrytí kódu spadá do techniky strukturálního testování neboli metoda bílé skříňky. Strukturální testování porovnává chování testovaného programu s očekávaným chováním a umožňuje tak brát v potaz možné nástrahy ve struktuře a logice, zatímco funkcionální porovnává chování s požadavky ve specifikaci bez jakéhokoliv ohledu na vnitřní fungování. Strukturálnímu testování se také říká testování cest, jelikož při tvorbě testovacích případů si volíme cestu strukturou programu [6][13]. Strukturální testování nám neumožňuje nalézt chyby způsobené vynecháním specifikace, tento nedostatek je ale ke konci vývoje zanedbatelný, jelikož se v pozdních fázích vývoje specifikace již málo mění a roli specifikace přebírá částečně produkt samotný.

Základní předpoklady stojící za analýzou pokrytí nám poukazují na výhody a nedostatky této testovací techniky. Mezi základní předpoklady patří eliminace nedosažitelného kódu, chyby, které jsou svázány s řídící strukturou, mohou být odhaleny tím, že tuto řídící strukturu změníme. Dalším předpokladem je schopnost hledat chyby bez vědomí, jaké chyby vůbec mohou nastat. Úspěšný průběh testů implikuje správnost programu [11]. Tester ví, jak by se měl program správně chovat a díky tomu je schopen identifikovat rozdíly od tohoto očekávaného chování. Samozřejmě ne vždy takovéto předpoklady platí. Analýza pokrytí je schopna odhalit nějaké možné chyby, není však ani zdaleka schopna odhalit všechny druhy chyb. Hodí se také spíše na aplikace, ve kterých probíhá spousta rozhodování, než na aplikace zaměřené na data jako databázové aplikace.

2.2 Základní metriky

2.2.1 Pokrytí příkazů

Tato metrika pozoruje, zda jsme při testování narazily na všechny spustitelné příkazy. Deklarativní příkazy, které generují spustitelný kód, jsou považovány za tyto spustitelné příkazy. Příkazy pro kontrolu vykonávání jako if, for a switch jsou pokryty jenom v případě, pokud je pokryt kontrolní výraz i se všemi výrazy, které obsahuje. Implicitní výrazy jako třeba vynechaný return nejsou do tohoto pokrytí zahrnuty. Tomuto pokrytí se také říká řádkové či segmentové pokrytí, nebo i pokrytí základních bloků, kde se za jednotku měření pokrytí považuje sekvence výrazů bez větvení [6], [11].

Hlavní výhodou této metriky je možnost ji aplikovat přímo na kód objektu bez nutnosti jej nějako předem zpracovat. Tato metrika se často využívá v profilerech výkonu. Naopak hlavní nevýhodou je neschopnost rozlišit některé kontrolní struktury. Například v následujícím kódu.

```
int* p = NULL;
if (condition)
    p = &variable;
*p = 123;
```

I bez testovacího případu, který by vyhodnocoval chování v případě, že bude condition rovna false bude při použití pokrytí příkazů vyhodnoceno pokrytí jako úplné. Ve skutečnosti pokud takový případ nastane a bude podmínka condition vyhodnocena jako false dojde k selhání kódu. Toto je jedna z největších slabin tohoto druhu pokrytí. Neurčí nám také, jestli bylo dosaženo ukončující podmínky v cyklu, pouze to že bylo vykonáno tělo cyklu. Toto může ovlivnit cykly obsahující break. Podmínku do-while navíc považuje na stejné úrovni jako nevětvený blok výrazů, jelikož se vždycky provede alespoň jedenkrát. Dále nerozlišuje, zda se v podmínkách vyskytují logické operátory || a &&, není také schopno rozlišit jednotlivé části switch výrazu [10].

Testovací případy se obecně zaměřují spíše na rozhodování než na příkazy. Není pravděpodobné, že bychom měli deset různých testů na sekvenci deseti po sobě jdoucích příkazů bez větvení, běžně by se všechno dalo do jednoho testovacího případu. Vezměme si jako příklad if-else výraz. Pokud bychom měli jeden příkaz za then a 99 za else, můžeme po vykonání jedné z možných cest získat extrémní výsledky buďto 1% nebo 99% pokrytí.

Ale musíme přiznat ve prospěch pokrytí příkazů, že oproti ostatním metrikám reflektuje procento pokrytých volání příkazů procento nalezených chyb za předpokladu, že jsou tyto chyby spíše svázány s kontrolou chodu kódu než s výpočty.

2.2.2 Pokrytí rozhodování

Tato metrika nám udává, zda byly boolean výrazy v řídících strukturách vyhodnoceny na oboje možnosti, jak true tak i false. Celý boolovský výraz je považován za jeden true-or-false predikát bez ohledu na obsah logických operátorů and a or. Navíc jsou v této metrice také zahrnuty pokrytí případů ve switch výrazu, zpracování výjimek a všechny body vstupu a výstupu. Konstantní výrazy kontrolující cestu jsou ignorovány [10], [11]. Nazývá se také pokrytí větví.

Někdy se pokrytí rozhodování a větví rozlišuje, pokrytí větví se považuje za slabší. Pokrytí rozhodování v tomto kontextu vyžaduje všechny boolovské podmínky ověřeny na true a false, i ty, které nijak neovlivňují kontrolu.

Tato metrika si zachovává všechny výhody jednoduchosti bez problémů pokrytí příkazů. Má však také svoji nevýhodu a ta spočívá v ignorování větví v boolovském rozhodování, které nastávají z důvodu zkráceného vyhodnocování těchto výrazů. Provedeme si ukázku na příkladu.

```
if (condition1 && (condition2 || function1()))
    statement1;
else
    statement2;
```

Tato metrika může vyhodnotit tuto strukturu jako zcela pokrytou i bez zavolání function1(). Toto může nastat v případě, že je condition1 i condition2 true, kde je celý výraz vyhodnocen jako true bez nutnosti vyhodnocovat function1() a taktéž pokud nabude condition1 hodnoty false vyhodnotí program celou podmínku jako false bez zjišťování dalších hodnot.

2.2.3 Pokrytí podmínek

Pokrytí podmínek nám udává true nebo false výstup každé podmínky. Za podmínku považujeme operand logického operátoru, který neobsahuje další logické operátory [11]. Dochází k měření podmínek nezávisle na sobě. Tato metrika je podobná pokrytí rozhodování, ale je citlivější na řídící struktury. Ale je nutné si uvědomit, že úplné pokrytí podmínek nám nezaručuje úplné pokrytí rozhodování.

```
bool f(bool e) { return false; }
bool a[2] = { false, false };
if (f(a && b)) ...
if (a[int(a && b)]) ...
if ((a && b) ? false : false) ...
```

Všechny tři if výrazy půjdou false větví bez ohledu na hodnoty a a b. Když ale využijeme všech možných kombinací hodnot proměnných a a b tak nám pokrytí podmínek vyhodnotí pokrytí jako úplné.

2.2.4 Více-podmínkové pokrytí

Více-podmínkové pokrytí nám udává, jestli nastaly všechny možné kombinace podmínek. Testovací případy pro plné pokrytí vycházejí z pravdivostní tabulky logických operátorů z rozhodovacího výrazu [11]. Pro jazyky se zkráceným vyhodnocováním výrazů je potřeba velmi důkladné otestování [6]. Pro tyto jazyky je více-podmínkové pokrytí velmi podobné podmínkovému pokrytí.

Hlavní nevýhodou je těžkost určení minimální sady testovacích případů, zvláště u komplexních výrazů. Navíc může být počet těchto testovacích případů značně rozdílný i pro podmínky s podobnou složitostí, jako například v následující ukázce.

```
a && b && (c || (d && e))
((a || b) && (c || d)) && e
```

```
   a && b && (c || (d && e))
1. F    -     -     -    -
2. T    F     -     -    -
3. T    T     F     F    -
4. T    T     F     T    F
5. T    T     F     T    T
6. T    T     T     -    -

    ((a || b) && (c || d)) && e
 1.   F    F       -    -      -
 2.   F    T       F    F      -
 3.   F    T       F    T      F
 4.   F    T       F    T      T
 5.   F    T       T    -      F
 6.   F    T       T    -      T
 7.   T    -       F    F      -
 8.   T    -       F    T      F
 9.   T    -       F    T      T
10.   T    -       T    -      F
11.   T    -       T    -      T
```

Jak lze vidět, pro dosažení úplného vice-podmínkového pokrytí stačí u prvního výrazu 6 testovacích případů, zatímco u druhého potřebujeme 11, což je téměř dvojnásobek. Navíc stejně jako u pokrytí podmínek nezahrnuje ani více-podmínkové pokrytí, pokrytí rozhodování. Pro jazyky bez zkráceného vyhodnocování představuje více-podmínkové pokrytí efektivně pokrytí cest pro logické výrazy.

2.2.5 Pokrytí cest

Tato metrika zaznamenává, zda se prošly všechny existující cesty v každé funkci. Cestou myslíme unikátní sekvenci větví od vstupu funkce až po její konec [11]. Také se může nazývat predikátovým pokrytím a na cestu je nahlíženo jako na možnou kombinaci logických podmínek.

Jelikož cykly představují velké množství možných cest, bere tato metrika v úvahu pouze omezený počet opakování. Pro řešení problému s cykly existuje spousta variací této metriky. Například hraničně vnitřní testování cest bere v potaz pouze dvě varianty a to nula opakování nebo více jak nula opakování, výjimkou je do-while cyklus kde jsou tyto dvě podmínky jedno nebo více než jedno opakování [11].

Pokrytí cest má výhodu v tom, že vyžaduje velmi důkladné testování. Má ovšem také dvě vážné nevýhody. První spočívá v počtu cest, který je exponenciální vůči počtu větví. Například funkce s deseti if výrazy má 1024 cest, které je potřeba otestovat a přidáním jediného if výrazu se tento počet opět zdvojnásobí na 2048. Další nevýhodou je to, že spoustu cest je nemožné vykonat z důvodu vzájemného vztahu dat.

```
if (success)
    statement1;
statement2;
if (success)
    statement3;
```

Pokrytí cest považuje tento fragment kódu za č možné cesty, ve skutečnosti jsou možné pouze dvě a to success = true a success = false. Pro vyřešení problému s velkým množstvím cest se vyvinulo množství variací pokrytí cest, jako například n-vzdálenostní pokrytí pod-cest, která nám udává, zda byly vykonány všechny cesty o délce n větví.

3. Možnosti Visual Studia 2010, frameworky a add-iny pro podporu testování.

3.1 Visual Studio Unit Testing Framework

Jedná se o sadu nástrojů integrovanou přímo do Visual Studia. Nabízí nám atributy pro určení testovacích tříd pomocí [TestClass] [16]. Takovéto třídy obsahují testovací metody. Dobrou praktikou je dávat do takovýchto tříd pouze unit testy. Dalším atributem, tentokrát pro testovací metody je [TestClass]. Toto identifikuje metody, které obsahují pouze testovací kód unit.

Assert je důležitou částí testovacího procesu, jelikož slouží k vyhodnocení testu. Je to kód, který porovnává podmínku či chování s očekávaným výsledkem. Ve Visual Studiu je k dispozici již připravená třída Assert.

Dále ještě máme inicializační a úklidové metody, které slouží k přípravě testovacího prostředí před spuštěním testů a následného uklizení po provedení testu. Inicializační metody jsou určeny atributem [TestInitialize] a úklidové metody [TestCleanup] [16].

Jako příklad si můžeme ukázat jednoduchý unit test, který nám poslouží jako ukázka zápisu atributů.

```
using Microsoft.VisualStudio.TestTools.UnitTesting;
using testTry;

namespace TestProject
{
    [TestClass]
    public class AdderUnitTest
    {
        [TestMethod]
        public void TestAdd()
        {
            Adder adderTestObject = new Adder();

            Assert.AreEqual(3, adderTestObject.Add(1, 2));
        }
    }
}
```

Visual Studio nám také nabízí možnost měření pokrytí kódu. Při spuštění předešlé ukázky testu dostaneme tyto výsledky.

Hierarchy	Not Covered (Blocks)	Not Covered (% Blocks)	Covered (Blocks)	Covered (% Blocks)
Vojta@VOJTA-NTB 2013-05-01 0...	7	43,75%	9	56,25%
TestProject.dll	0	0,00%	4	100,00%
testTry.exe	7	58,33%	5	41,67%
{} testTry	7	58,33%	5	41,67%
Adder	2	28,57%	5	71,43%
Add(int32,int32)	2	28,57%	5	71,43%

Nás zajímá v této chvíli poslední řádek, jelikož jsme testovali metodu add z třídy Adder. Máme také možnost si zobrazit, které části kódu jsme v testu vynechali a které jsme prošli. Následuje ukázka takového zobrazení, ve které zjistíme, proč nedošlo k úplnému pokrytí.

```
public class Adder
{
    public int Add(int x, int y)
    {
        if (x >= 0 && y >= 0)
        {
            return x + y;
        }
        else
        {
            return -1;
        }

    }
}
```

Z kódu je patrné, že sčítáme pouze v případě, že jsou obě čísla kladná. Lze si všimnou tří rozdílných barev. V testu jsme provedli assert na obě čísla kladná, tím byla splněna podmínka a proběhla část kódu ohraničená zelenou barvou. Metodu Add jsme v testu se zápornou hodnotou nevolali, tudíž je část za else ohraničena červenou a jedná se o jeden ze dvou nepokrytých bloků, které nám hlásilo zobrazení pokrytí kódu. Poslední barvou zůstává oranžová a ta nám značí, že jsme neotestovali všechny možné kombinace podmínky, a proto se taktéž počítá jako nepokrytý blok. Z tohoto je patrné, že se jedná o pokrytí větví.

3.2 NUnit framework

Jedná se o volně dostupný framework. Stejně jako framework visual studia nám nabízí různé funkce určené atributy. Pro určení testovací třídy slouží atribut [TestFixture], dále máme atributy [TestFixtureSetUp] a [TestFixtureTearDown], které mají stejnou funkci jako atributy [TestInitialize] a [TestCleanup] z frameworku Visual Studia, stejně tak [Test] je určení testovací metody, které již známe [17].

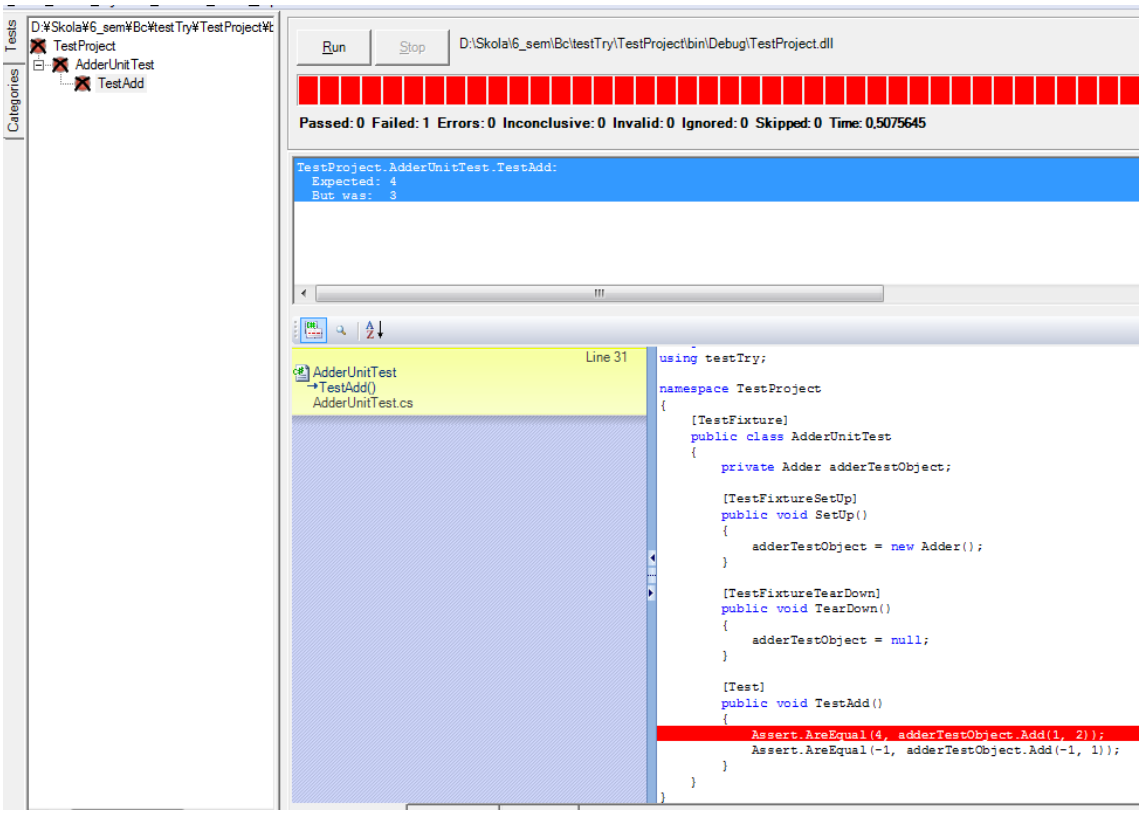
NUnit nám však také přidává pár nových atributů a jejich možností. Například [Test, ExpectedException(typeof(<<ExceptionType>>))] [17]. Takovýto typ testu neobsahuje volání třídy Assert a druh výjimky musí být přesně určen. Kdybychom <<ExceptionType>> nahradili obecným Exception skončil by test neúspěšně. Nabízí nám samozřejmě ještě spoustu dalších možností pro opakované spouštění s rozdílnými hodnotami atributů, náhodnými čísly a další.

NUnit také obsahuje vlastní GUI pro spouštění NUnit testů. Po úpravě testu na třídu Adder pro potřeby NUnitu lze takovýto test spustit v GUI pro NUnit.

```
using NUnit.Framework;
using testTry;

namespace TestProject
{
    [TestFixture]
    public class AdderUnitTest
    {
        private Adder adderTestObject;

        [TestFixtureSetUp]
        public void SetUp()
        {
            adderTestObject = new Adder();
        }

        [TestFixtureTearDown]
        public void TearDown()
        {
            adderTestObject = null;
        }

        [Test]
        public void TestAdd()
        {
            Assert.AreEqual(3, adderTestObject.Add(1, 2));
            Assert.AreEqual(-1, adderTestObject.Add(-1, 1));
        }
    }
}
```

Takto upravený test obsahující správné atributy a hlavně využívající NUnit framework což je patrné z using NUnit.Framework můžeme následně spustit.

Po spuštění je jasné, že test prošel úspěšně. V případě, že test neuspěje lze dobře vidět která část testu selhala, i když máme více assertů v jedné testovací metodě, i když by se podle správných testovacích návyků měl dávat pouze jeden assert na testovací metodu.

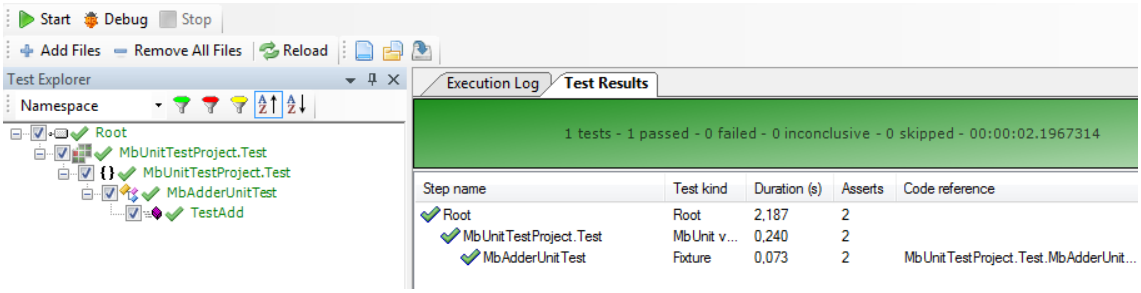


Z obrázku je patrné, že selhala testovací metoda TestAdd() v testu AdderUnitTest a to na řádku ohraničeném červenou barvou.

Bohužel nám tento nástroj jako takový nenabízí žádné možnosti měření pokrytí kódu testy.

3.3 MbUnit framework

MbUnit framework je hodně podobný NUnit frameworku. Nabízí podobné možnosti atributů, zápis samotný je taktéž hodně podobný. Například atributy [TestFixture] a [Test] jsou stejné, taktéž [TestFixtureSetUp] a [TestFixtureTearDown], můžeme ale používat kratší zápis [SetUp] a [TearDown] [18]. Nabízí také spoustu dalších možností pro testování více-vláknových aplikací, zvládá stejně jako NUnit očekávané výjimky a další možnosti. Taktéž jako u NUnit frameworku, musíme použít using MbUnit.Framework; pro import všeho potřebného a následně je tyto testy možno spustit v aplikaci Galio Icarus, která slouží podobně jako GUI u NUnit frameworku. V následující ukázce byl spuštěn stejný test, jako v předcházejících případech, tudíž bude samozřejmě úspěšný.



Vidíme z toho, které testy byly vykonány a s jakým výsledkem, v pravé části je kromě trvání testu také počet assertů v testu. Bohužel stejně jako u NUnit nám toto prostředí nenabízí žádnou možnost měření pokrytí kódu.

3.4 TestDriven.NET

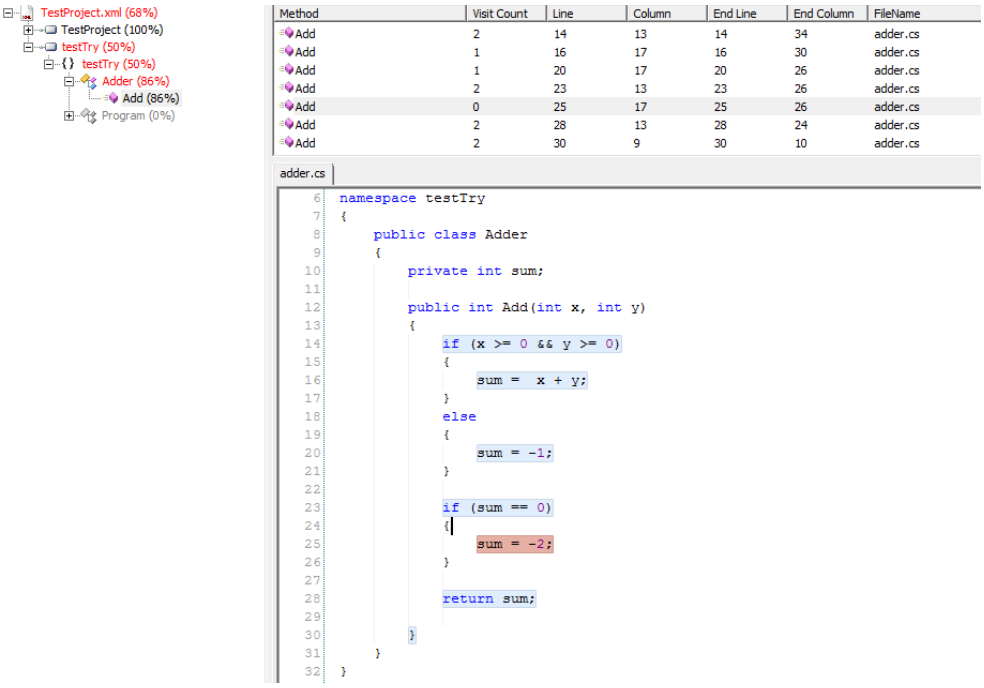
Jedná se o add-in pro Visual Studio, a slouží ke spouštění unit testů. Podporuje vícero frameworků, včetně NUnit a MbUnit. Díky tomuto nástroji lze snadno spouštět unit testy těchto frameworků přímo v prostředí Visual Studia. Při spuštění testu se automaticky zjistí framework, který byl použit a spustí test s použitím správného testovacího spouštěče. Při spuštění testu, který jsme použili pro framework NUnit vypadá výstup následovně

```
------ Test started: Assembly: TestProject.dll ------

1 passed, 0 failed, 0 skipped, took 0,65 seconds (NUnit 2.6.1).
```

Při úspěchu moc informací nezískáme, což samozřejmě není potřeba, při výskytu chyby už je však výřečnější.

```
------ Test started: Assembly: TestProject.dll ------

Test 'TestProject.AdderUnitTest.TestAdd' failed:
  Expected: 4
  But was:  3
    AdderUnitTest.cs(31,0): at TestProject.AdderUnitTest.TestAdd()

0 passed, 1 failed, 0 skipped, took 5,52 seconds (NUnit 2.6.1).
```

Zobrazí se nám přesně, který test selhal, co jsme dostali a co jsme očekávali. Nicméně co nás hlavně bude zajímat na tomto nástroji je, že obsahuje nástroj NCover, který nám umožní měřit pokrytí kódu s využitím unit testů NUnit a MbUnit frameworku. Ukážeme si opět příklad na NUnit testu.



V levé části lze vidět procentuální pokrytí, zeleně a červeně rozlišená pokrytá a nepokrytá místa v kódu a taktéž počet vykonání jednotlivých řádků. Druhou možností jak měřit pokrytí kódu je pomocí nástroje Coverage, který umožní zobrazit pokrytí kódu v Test Exploreru Visual Studia.

3.5 Shrnutí

Visual Stuio Unit Testing Framework – nabízí základní možnosti unit testů, jako nastavení před provedením testu a uklizení po testu, třídu Assert a v Test Exploreru možnost zobrazení pokrytí kódu [16].

NUnit Framework – nabízí kromě základních možností i pokročilé atributy pro testování s náhodnými hodnotami či rozsahy hodnot a opakovaného spouštění pro všechny hodnoty rozsahu. Dále můžeme očekávat, že testovací metoda vrátí výjimku a můžeme také tvořit skupiny testů [17]. Nenabízí žádnou možnost měření pokrytí kódu, jako tomu bylo u VS frameworku.

MbUnit Framework – nabízí podobné možnosti jako NUnit framework, navíc však má i možnosti pro testování více-vláknových aplikací, vícenásobný assert a generování náhodných stringů [18]. Stejně jako NUnit framework nenabízí možnost měření pokrytí kódu.

TestDrive.NET – Tento nástroj slouží ke spouštění unit testů jiných frameworků v prostředí Visual Studia a s pomoci NCover či Coverage nám nabízí také možnost měřit jejich pokrytí.

4. Postup testování zvoleného modulu

Pro testování využijeme MbUnit Framework a NCover pro zjištění pokrytí. NCover nám nabízí pokrytí větví. Před testováním je také nutné si zvolit procentuální hranici, které chceme dosáhnout při testování. Pro naše účely zvolíme hranici 90%.

Nejdříve si vytvoříme nový testovací projekt, při použití MbUnit nevyužíváme standardního testovacího projetu, ale vytvoříme MbUnit v3 Test Project. Toto nám zajistí všechny potřebné reference. Poté vytvoříme prázdný unit test. Nyní zaměníme v souboru unit testu using Microsoft.VisualStudio.TestTools.UnitTesting za using MbUnit.Framework.

Další etapou jsou potřebné nastavení pomocí atributů SetUp a TearDown. V těchto atributech dojde k nastavení testovací databáze a k následnému navrácení do původního stavu po skončení testu.

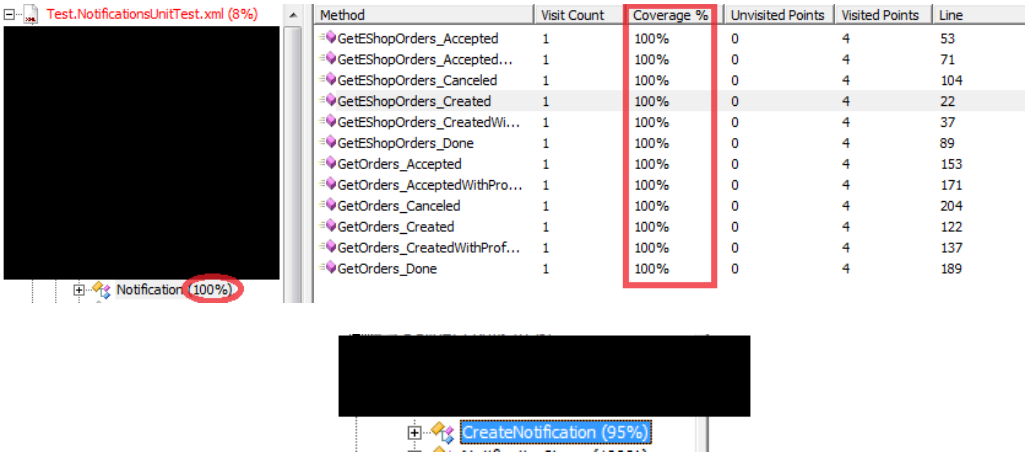
```
private String originalConnectionString;
private String testConnectionString = @"Data Source=VOJTA-NTB\SQLEXPRESS;Initial Catalog=Modacar_ne

[SetUp]
public void SetUp()
{
    originalConnectionString = Configuration.Config.ConnectionString;
    Configuration.Config.ConnectionString = testConnectionString;
}

[TearDown]
public void TearDown()
{
    Configuration.Config.ConnectionString = originalConnectionString;
}
```

Po nastavení těchto věcí se můžeme pustit do testování jednotlivých metod. Toto probíhá v testovacích metodách označených atributem Test.

```
[Test]
public void TestGetEShopOrders_Created()
{
    int[] lActual;
    int[] lExpected;
    lActual = Notification.GetEShopOrders_Created();

    using (IS.DAL.DataContext lDataContext = new IS.DAL.DataContext(false, false, false))
    {
        lExpected = lDataContext.Documents.Where(aR =>
                aR.Type == (int)DocumentType.ReceivedOrderEShop &&
                aR.State == (int)DocumentStateReceivedOrder.S_00_New
                && !aR.DocumentNotifications.Any(aN =>
                    aN.DocumentTemplatePredefined == (int)DocumentTemplatePredefined.EShopOrderCreated &&
                    aR.ID == aN.DocumentID))
                .Select(aR => aR.ID).ToArray();
    }

    Assert.AreElementsEqualIgnoringOrder(lExpected, lActual);
}
```

V testovací metodě si připravíme dvě proměnné lActual, která bude obsahovat výstup testované metody a lExpected, do které dáme očekávaný výsledek. Jelikož v tomto případě porovnáváme dvě pole, využijeme u třídy Assert funkce AreElementsEqualIgnoringOrder, jelikož nám

i nezáleží na pořadí prvků. Tento postup se opakuje pro všechny public metody, jelikož unit testování slouží pro testování veřejného interface tříd.



Pro provedení všech testů jsme dosáhli 100% a 95% pokrytí na jednotlivých částech modulu, tudíž jsme splnili námi určenou podmínku. Běžně není snadné dosáhnout 100% pokrytí, jelikož by se takové úsilí nevyplatilo. Jelikož jsme tentokrát testovali pouze jeden modul, nebylo u něj dosažení takového pokrytí zase tak obtížné.

Závěr

Cílem této bakalářské práce bylo seznámit se s testováním a jeho druhy, různými metrikami pokrytí kódu a přiblížit si možnosti unit testování na platformě .NET. Hlavním důvodem tohoto snažení je narůstající důraz na kvalitu a taktéž částečného předcházení rizik spojených s vývojem software, který by byl následně odmítnut z důvodů nedostatečné kvality a chybovosti.

V teoretické části byla představena většina nejdůležitějších druhů testování, které při kombinaci vícero druhů zaručují vysokou kvalitu výsledného software a prevenci chyb, které by bylo později velmi obtížné a nákladné odstranit. Znalost těchto testů a jejich společné využívání je z mého pohledu velmi důležité jak z hlediska kvality, tak z hlediska ekonomického.

Dále byly v teoretické části zahrnuty taktéž metriky pokrytí kódu testy. Tyto metriky jsou přímo provázány s testy a určují nám, o jaké části software jsme schopni prohlásit, že funguje, jak očekáváme. Taktéž z nich lze vyvodit ekonomické závěry, jelikož lze tyto metriky úspěšně použít jako doklad míry postupu v projektu pro management a zákazníka a ujistit tak sebe i objednavatele software o plnění termínu.

Praktická část byla zaměřena na unit testy na patformě .NET, možnosti Visual Studia 2010 pro tvorbu unit testů a měření pokrytí kódu a taktéž dalších volně dostupných frameworků, ze kterých jsme si představili NUnit a MbUnit, nabízející značné možnosti a ve spojení s add-iny pro vývojové prostředí Visual Studia a umožňující integraci a využití jeho částí jako měření pokrytí. NUnit i MbUnit nabízejí v porovnání s Visual Studio Unit Test Frameworkem daleko více možností, ale postrádají možnost měření metrik. Tento nedostatek jsme ovšem byly schopni doplnit nástrojem TestDriven.NET, který umožňuje nejenom spouštět testy těchto frameworků přímo z vývojového prostředí Visual Studia, ale taktéž nám nabízí integraci s měřením pokrytí Visual Studia, anebo jiný nástroj NCover, který taktéž slouží pro měření pokrytí. Při navržené kombinaci frameworku MbUnit a jeho možností spojených s nástrojem NCover pro měření metrik lze snadno tvořit unit testy podle pospaného postupu.

Z hlediska dalšího vývoje by bylo ideální rozšířit tyto nástroje a metodiky i o další typy testů, ne jenom unit testy a vytvořit tak úplnou metodologii pro testování na platformě .NET s popisem nástrojů, jejich možnostmi a hlavně různými kombinacemi tak, aby byla zaručena co nejsnazší tvorba testů a sjednoceny postupy.

Literatura

[1] KANER, Cem, Jack L FALK a Hung Quoc NGUYEN. *Testing computer software*. 2nd ed. New York: John Wiley, 1999, xv, 480 s. ISBN 04-713-5846-0.

[2] BEIZER, Boris, Jack L FALK a Hung Quoc NGUYEN. *Black-box testing: techniques for functional testing of software and systems*. 2nd ed. New York: Wiley, c1995, xxv, 294 p. ISBN 04-711-2094-4.

[3] HUIZINGA, Dorota a Adam KOLAWA. *Automated defect prevention: best practices in software management*. Hoboken, N.J.: IEEE Computer Society, c2007, xxvi, 426 p. ISBN 04-700-4212-5.

[4] MCCONNELL, Steve a Adam KOLAWA. *Code complete: best practices in software management*. 2nd ed. Washington: Microsoft Press, c2004, xxxvii, 914 s. ISBN 07-356-1967-0.

[5] MYERS, Glenford J a Adam KOLAWA. *The art of software testing: best practices in software management*. 2nd ed. New York: Wiley, c2004, xi, 177 p. ISBN 04-710-4328-1.

[6] BEIZER, Boris a Adam KOLAWA. *Software testing techniques: best practices in software management*. 2nd ed. New York: Van Nostrand Reinhold, c1990, xxvi, 550 p. ISBN 04-422-0672-0.

[7] DUSTIN, Elfriede a Adam KOLAWA. *Effective software testing: 50 specific ways to improve your testing*. 2nd ed. Boston: Addison-Wesley, 2002, xv, 271 p. ISBN 02-017-9429-2.

[8] BINDER, Robert a Adam KOLAWA. *Testing object-oriented systems: models, patterns, and tools*. 3rd ed. Reading, Mass.: Addison-Wesley, c2000, xlviii, 1191 p. ISBN 02-018-0938-9.

[9] BLACK, Rex a Adam KOLAWA. *Managing the testing process: practical tools and techniques for managing software and hardware testing*. 3rd ed. Indianapolis, IN: Wiley, 2009, xlviii, 1191 p. ISBN 04-704-0415-9.

[10] BLACK, Rex a Adam KOLAWA. *IEEE Transactions on software engineering: practical tools and techniques for managing software and hardware testing*. 3rd ed. New York: Institute of Electrical and Electronics Engineers, Inc, 2009, xlviii, 1191 p. ISBN 0098-5589.

[11] PEZZÈ, Mauro a Michal YOUNG. *Software testing and analysis: process, principles, and techniques*. 3rd ed. Hoboken: John Wiley, 2008, xxii, 488 s. ISBN 978-0-471-45593-6.

[12] PATTON, Ron. *Software testing*. 2nd ed. Indianapolis: Sams Publishing, 2006, xiv, 389 s. ISBN 06-723-2798-8.

[13] MCCAFFREY, James. *Software testing: fundamental principles and essential knowledge*. 2nd ed. [Lexington, KY: www.booksurge.com], 2010, xiv, 389 s. ISBN 14-392-2907-4.

[14] SUTTON, Michael, Adam GREENE a Pedram AMINI. *Fuzzing: brute force vulnerabilty discovery*. 2nd ed. Upper Saddle River, NJ: Addison-Wesley, c2007, xxvii, 543 p. ISBN 978-032-1446-114.

[15] TAKANEN, Ari, Jared D DEMOTT a Charles MILLER. *Fuzzing for software security testing and quality assurance: brute force vulnerabilty discovery*. 2nd ed. Norwood, MA: Artech House, c2008, xxii, 287 p. ISBN 978-159-6932-142.

[16] MICROSOFT. *Verifying Code by Using Unit Tests* [online]. 2013 [cit. 2013-04-15]. Dostupné z: http://msdn.microsoft.com/en-us/library/dd264975(v=vs.100).aspx

[17] NUNIT. *NUnit 2.6.2* [online]. 2012 [cit. 2013-04-15]. Dostupné z: http://www.nunit.org/index.php?p=docHome&r=2.6.2

[18] MBUNIT. *MbUnit online documentation* [online]. 2012 [cit. 2013-04-15]. Dostupné z: http://www.mbunit.com/

Přílohy

Příloha na DVD.

Obsahuje Visual Studio 2010 projekt, ve kterém jsou celé vytvořené unit testy a testované třídy s ostatními potřebnými třídami s prázdnými metodami a pouze využitými atributy z důvodu veřejné nepřístupnosti testovaného zdrojového kódu.